Network Programming Work
Request-Reply Messaging App

The aim of this work is to become familiar with network programming using Sockets, Input / Output streams and Threads (or RMI) in order to implement programs based on the client-server model. As part of this work, a distributed messaging system using a simple request-reply protocol will be developed. Clients send to

Server a Request and then the Server responds with a Response and their connection terminates.

In the system different users will be able to create accounts and send messages to each other. This functionality will be provided by:

a) a server program, which will have the ability to manage multiple requests from clients simultaneously.

b) client programs, each of which will have the ability to send requests to the server.

The main elements of the program are presented below.

Message

Each message sent, received, or stored on the server must have the following fields:

Capacity	**Description**
boolean isRead	Indicates if the message has already been read.
String sender	The sender of the message.
String receiver	The recipient of the message.
String body	The text of the message.

You can complete any other property you deem necessary for the implementation of the system.

Account

Each user account will be stored on the server in the following format:

Capacity	Description
String username	The username. It consists only of alphanumerics and the special character "_".
Int authToken	A unique user identification number (created by the server and is personal / hidden).
List <Message> messageBox	The user's mailbox, which is a list of Messages.

You can complete any other property you deem necessary for the implementation of the system.

MessagingClient

The client communicates with the server in the client program.

The program will receive input from the user by sending it to the server, while at the same time it will receive data from the server and display it appropriately to the user.

MessagingServer

The server will run continuously as a service "listening" to incoming requests from customers. Each incoming request will be assigned to a different thread (you can use Threads or the RMI mechanism), so that multiple requests can be served at the same time. Also, there will be a list of accounts (Account) on the server, so that data such as registered users, their passwords and their mailboxes are kept. Finally, methods for communicating with the user must be implemented.

Functions

The delivered programs must definitely perform the functions that we will describe below. Executable programs must give

results **exactly in the format we describe in the Scenarios tables.**

Server

The server should run as follows:

java server <port number>

where Port Number is the door to which you will hear requests.

Client

Client is the user interface with the program. The general execution command is:

java client <ip> <port number> <FN_ID> <args>

where

Ip ip: The IP address of the Server
• port number: The port on which the Server is listening
• FN_ID: The ID of the function to be executed • args: the parameters of the function

The following functions are provided:

Create Account (FN_ID: 1) java client <ip> <port number> 1 <username>

Creates an account for the user and uses the given username. The function returns a unique code (token) which is used to authenticate the user in his next requests.

Scenario	**Printing**
Success	<integer>
Example: $> java client localhost 5000 1 tester 1024 In the above example the tester user, from now on, will use as auth Token the number 1024 in his next requests.	
The user already exists	Sorry, the user already exists
Example: $> java client localhost 5000 1 tester Sorry, the user already exists	
Wrong username format	Invalid Username
Example	

$> java client localhost 5000 1 inv @ -lid Invalid Username

Show Accounts (FN_ID: 2)
java client <ip> <port number> 2 <authToken>

Shows a list of all accounts in the system.

Scenario	**Printing**
In all cases it prints the user list in the format given	1. <username_1> 2. <username_2> … n. <username_n>
Example: $> java client localhost 5000 2 1024 1. demo 2. raven_13 3. dr4g0n_sl4y3r	

Send Message (FN_ID: 3)
java client <ip> <port number> 3 <authToken> <recipient>
<message_body> Sends a message (<message_body>) to the account with username <recipient>.

Scenario	**Printing**
Successful mission	OK
Example: $> java client localhost 5000 3 1024 tester “HELLO WORLD” OK	
If the <recipient> user profile does not exist	User does not exist
Example: $> java client localhost 5000 3 1024 friend “HELLO WORLD” User does not exist	

Show Inbox (FN_ID: 4) java
client <ip> <port number> 4 <authToken>

Displays the list of all messages for a specific user.

Displays a list of all messages in the user's messagebox. The format to print them is as follows:

Scenario	**Printing**
Success	<message_id_1>. from: <username_x> *? <message_id_2>. from: <username_y> *? … <message_id_n>. from: <username_z> *?
The asterisk will only be visible if this message has not been read. Example: $> java client localhost 5000 4 1024 27. from: raven_13 * 43. from: demo * 55. from: raven_13 * 58. from: raven_13 * 67. from: dr4g0n_sl4y3r In our example the user message dr4g0n_sl4y3r has already been read.	

ReadMessage (FN_ID: 5) java
client <ip> <port number> 5 <authToken> <message_id> This function displays the contents of a user's message with id <message_id>. The message is then marked as read.

If there is a message the program prints the following

Scenario	**Printing**
Success	(<sender>) <message>
Example: $> java client localhost 5000 5 1024 43 (demo) Hello World	
The message does not exist	Message ID does not exist
Example: $> java client localhost 5000 5 1024 1111 Message ID does not exist	

DeleteMessage (FN_ID: 6) java
client <ip> <port number> 6 <authToken> <message_id> This function deletes the message with id <message_id>.

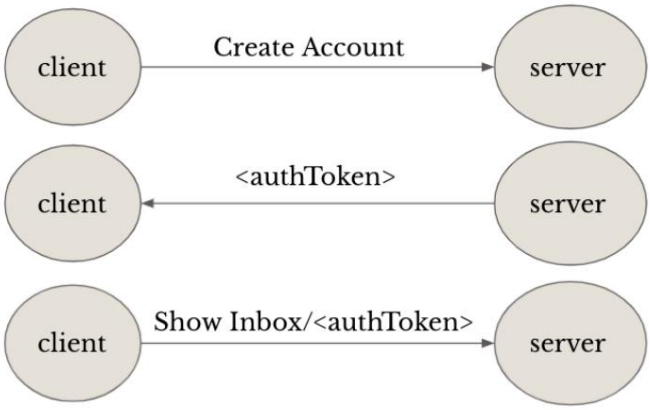
Scenario	**Printing**
Success	OK
Example: $> java client localhost 5000 5 1024 43 OK	
The message does not exist	Message does not exist
Example: $> java client localhost 5000 5 1024 1111 Message does not exist	

Auth Token

In general, since the communication protocol that will be created does not retain data for the user in each session, we must ensure that:

- No user can read another user's data. . No user can send messages as another user.

For this reason we use the Auth Token in every Request. The Server must associate each auth token with an Account (when creating the account).



In case an Auth Token is given incorrectly (ie authToken is not assigned to a user) all functions must return the corresponding message.

Scenario	**Printing**
Wrong authToken	*Invalid Auth Token*

Deliverable

Deliver a zip file in Latin characters and format:

<aem> _ <name> _ <surname> .zip

e.g. 1243_petros_riginos.zip

which contains the following:

• src /: Source files. Ars jars /: ient
Client.jar: Client Execution File •
Server.jar: Server Execution File •
README.md A text file that will briefly
describe the
classes you implemented and the assumptions you made about implementing the system.

Attention: It is important that the above is **submitted strictly in the correct format and file structure** in order to complete the submission process smoothly. We recommend that you run your programs as jar files to confirm that they are working properly before sending them.

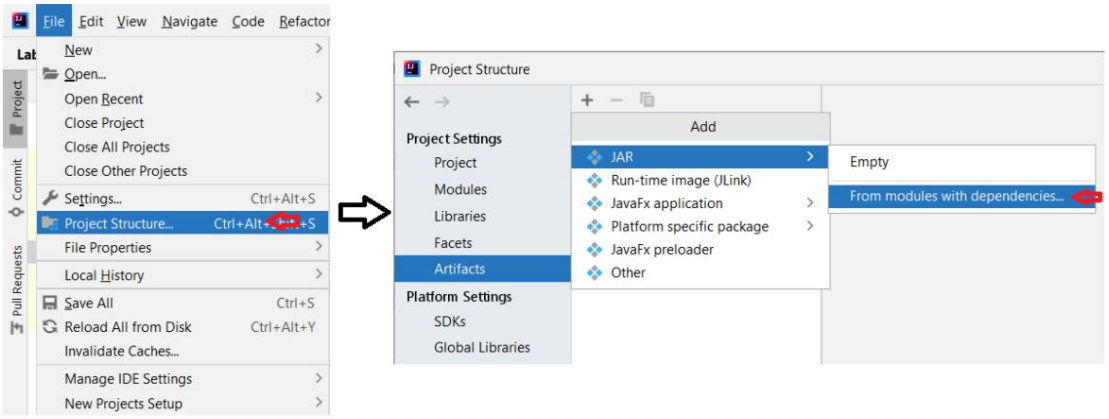
java -jar Server.jar 5000 java -jar
Client.jar localhost 5000 1 demousername

Creating an Executable Jar (IntelliJ):

Step 1:

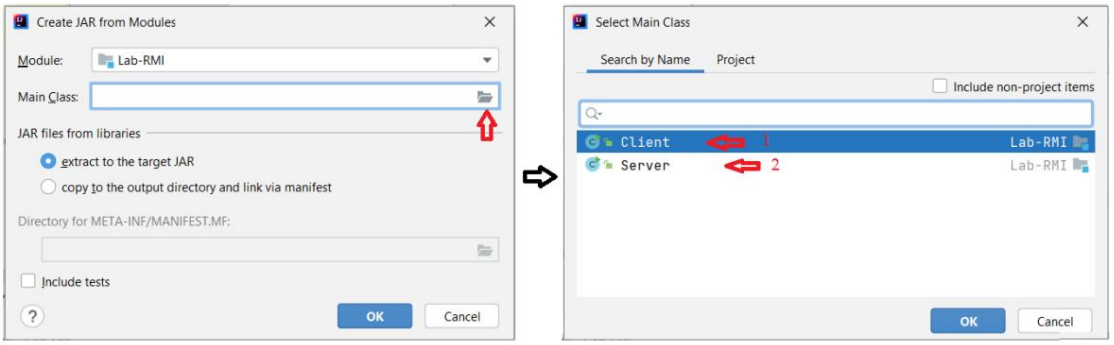
We follow the route…

File> Project Structure> Project Settings> Artifacts> Click green plus sign> Jar> From modules with dependencies…

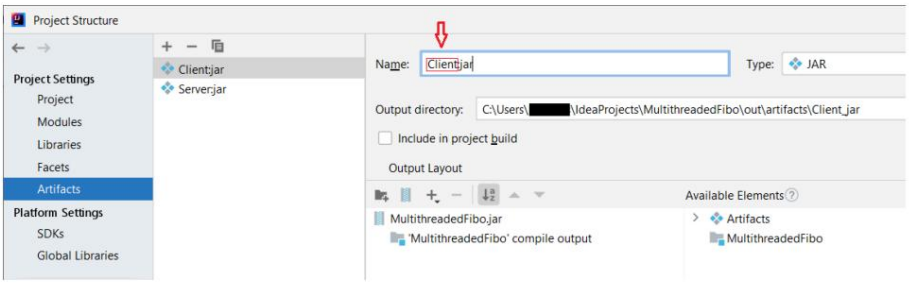


Step 2:

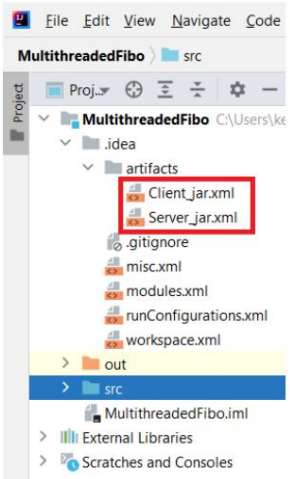
Even if we have a project with two files that have the main method, that means we have to create two .jar files. The second image below shows these two project files eg Client.java and Server.java. Arrows 1 and 2 indicate that the .jar file creation process should be repeated for both.



For each project title. We do this to know in the end in which folders the final .jar files will be produced for each one.

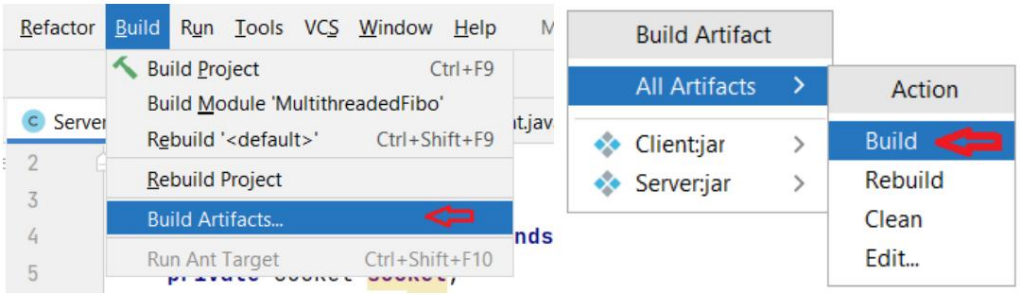


The project navigator will then display two xml files as shown in the following image in the red box:



Step 3:

Finally, we create the executable compressed .jar as shown in the image below.



Intellij will store the final two .jar files in the project out folder in two sub-folders Client_jar and Server_jar, respectively. Intellij gives the same name (project name dld <projectName> .jar) to both. However because they have to be delivered in a jars folder - as requested by the deliverables, you have to pick them up and rename them (right-click> refactor> rename> (proceed anyway)) to Client.jar and Server.jar when you will save them in the deliverable folder which will eventually become a zip file.